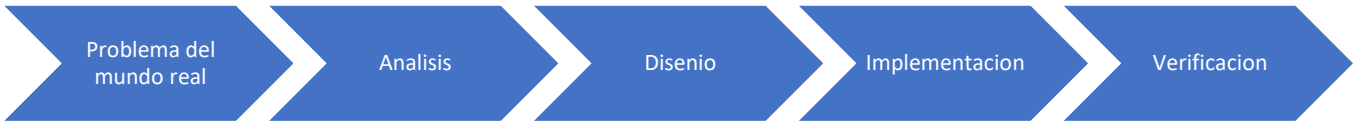
CONCEPTO DE ALGOTIRMOS, DATOS Y PROBLEMAS

# Introducción

La informática es una ciencia que estudia el análisis y resolución de problemas utilizando computadoras.

- Ciencia: relacionado con una metodología fundamentada y racional para el estudio y resolución de problemas.
- Resolución de problemas: aplicaciones en áreas diferentes como biología, comercio, control industrial, administración, etc.
- Computadora: maquina digital y sincrónica, con capacidad de cálculo numérico y lógico y de comunicación con el mundo exterior. Ayuda al compre a hacer tareas repetitivas en < tiempo y con > exactitud.

Para solucionar problemas del mundo real con una computadora, es necesario atravesar una serie de etapas:




1. Análisis: se analiza el problema en su contacto del mundo real. Deben obtenerse los requerimientos del usuario. El resultado es un modelo preciso del ambiente del problema y el objetivo a resolver. Se simplifican los requerimientos del problema y el contexto y datos a usar por el programa en la computadora.
2. Diseño: la descomposición funcional nos ayudara a reducir la complejidad, a distribuir el trabajo y en el futuro a reutilizar los módulos. Esta etapa involucra la especificación de algoritmos. Cada uno de los módulos diseñados tienen una función que podemos traducir en un algoritmo. La elección del algoritmo adecuado es muy importante para la eficiencia del sist. de software.
3. Involucra escribir los programas, los algoritmos escritos en la etapa de diseño se convierten en programas escritos e un lenguaje de programación concreto.
4. Una vez que se tienen los programas escritos y depurados de errores de sintaxis, se debe verificar que su ejecución conduce al resultado deseado usando datos representativos del problema real.

La etapa de análisis y diseño NO dependen del lenguaje de programación, en cambio la etapa de implementación sí.

# Concepto de algoritmo

Un algoritmo es una especificación rigurosa de la secuencia de pasos a realizar sobre un autómata para alcanzar un resultado en tiempo finito.

- Especificación rigurosa: debemos expresarlo en forma clara y univoca
- Tiempo finito: el algoritmo comienza y termina.

CONCEPTO DE ALGOTIRMOS, DATOS Y PROBLEMAS

## Concepto de programa

Programa: conjunto de instrucciones u órdenes ejecutables sobre una computadora, que permite cumplir con una función específica cuyas órdenes se expresan en un lenguaje de programación concreto.

Por lo tanto, escribir un programa exige:

- Elegir la representación adecuada de los datos del problema
- Elegir un lenguaje de programación según el problema y maquina a utilizar
- Definir el conjunto de instrucciones cuya ejecución ordenada conduce a la solución

PROGRAMA= INSTRUCCIONES + DATOS

## Programación imperativa

El modelo que siguen los lenguajes de programación para definir y operar la información, permite asociarlos a un paradigma de programación en particular.

Pascal es un lenguaje de paradigma imperativo.
PROGRAMA= INSTRUCCIONES + DATOS

INSTRUCCIONES: operaciones que ejecutara la computadora al interpretar el programa (escritos en un lenguaje determinado).

DATOS: valores de información que se necesita tener y/o transformar para cumplir la función del programa (cada lenguaje tiene los propios).

## Concepto de dato

DATO= representación de un objeto del mundo real. Los datos permiten modelizar aspectos del problema que se quieren resolver mediante un programa ejecutable en una computadora.

Tenemos dos tipos de datos:

CONSTANTES: datos los cuales no cambian durante la ejecución del programa

VARIABLES: datos que cambian durante la ejecución del programa

## TIPO DE DATO

es una clase de objetos de datos ligados a un conjunto de operaciones para crearlos y manipularlos.

Cada tipo de dato se caracteriza por presentar: un rango de valores posibles, conjunto de operaciones realizables sobre ese tipo, una representación interna.

Un aspecto muy importante en los lenguajes de programación es la capacidad de especificar y manejar datos no estándar, lo que permite: **aumentar la riqueza** expresiva del lenguaje (con mejores posibilidades de abstracción* de datos), **mayor seguridad** respecto de las operaciones que se realizan sobre cada clase de datos y establecer **límites** sobre los valores posibles que pueden tomar las variables que corresponden al tipo de dato.

**CLASIFICACION DE DATOS:**

**SIMPLES**: aquellos que toman un único valor en un momento determinado

DEFINIDOS POR EL LENGUAJE: provistos por el lenguaje. Su representación, operaciones y valores son reservados al mismo: numérico, carácter, lógico.

- Tipo numérico: permite representar el conjunto de los valores numéricos referidos a números enteros o reales.

El tipo de dato **entero** es un tipo de dato simple y ordinar que pesa 2 bytes.

El tipo de dato **real** permite representar números decimales. Es un tipo de dato simple. Pesa 6 bytes.

Operaciones del tipo de dato numérico: +, -, *, división real */,* división entera *div*, módulo *mod*, operación de asignación:= y operaciones de comparación *= <> < <= =>.*

- Tipo lógico: permite representar datos que pueden tomar solamente uno de los valores. Puede conocerse también como tipo de dato boolean. Es un tipo de dato simple y ordinal. Ocupa 1 byte

Operaciones permitidas del tipo de dato lógico: asignación *:=,* negación *not*, conjunción *and*, disyunción *or*.

- Tipo carácter: tipo de dato simple y ordinal. Ocupa 1 byte.

Operaciones permitidas del tipo de dato carácter: asignación *:=,* comparación *> => < <=.*

- Tipo puntero*

DEFINIDOS POR EL USUARIO: es aquel que no existe en la definición del lenguaje y el programador se encarga de su especificación. Permite definir nuevos tipos de datos a partir de datos primitivos.

Siempre tiene asociado: un rango de valore posibles, una forma de representación, operaciones permitidas, conjunto de condiciones de valores permitidos que se pueden verificar.

Tiene determinadas ventajas declarar estos tipos de datos que son:

- *Flexibilidad*: en el caso de ser necesario modificar la forma en que se representa el dato, solo se debe modificar una declaración en lugar de un conjunto de declaraciones de variables.
- *Documentación*: el uso de nombres auto explicativos como identificadores facilita el entendimiento y lectura del programa.
- *Seguridad*: reducen los errores por el uso de operaciones inadecuadas y se pueden obtener programas más confiables.

DATO SUBRNGO DEFINIDO POR EL USUARIO: tipo simple y ordinal que consiste de una sucesión de valores extraídos de un ordinal base. Los ordinales base permitidos son los enteros y caracteres. La ocupación en memoria se condiciona por el tipo base. Y sus operaciones se heredan del tipo base.

Facilitan el chequeo de posibles errores ya que permite al lenguaje verificar si los valores asignados están en el rango establecido y ayudan al mantenimiento del programa.

**Compuestos:** tipo de dato el cual puede tomar varios valores a la vez que guardan una relación lógica entre ellos.

CONCEPTO DE ALGOTIRMOS, DATOS Y PROBLEMAS

DATO COMPUESTO DEFINIDO POR EL USUARIO *STRING*: sucesión de caracteres de longitud determinada. La longitud es el número máximo de char que puede contener el dato. Si no especifico, en PASCAL el máximo es de 255 char.

La ocupación en memoria se determina por la longitud de la cadena +1 byte que almacena la cantidad de caracteres de la secuencia.

Operaciones permitidas: asignación *:=,* entrada/salida *read/write*, relación < <= = > =>

*DATO COMPUESTO DEFINIDO POR EL USUARIO RECORD **

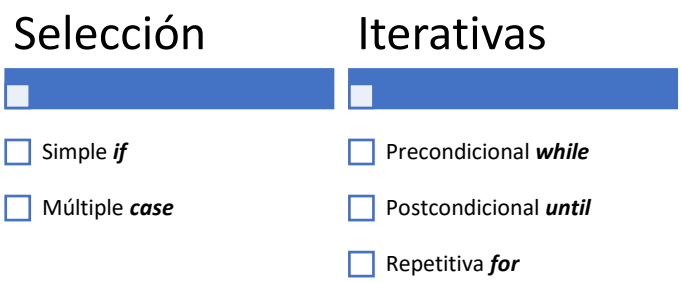
## Concepto de estructura de control

Todos los lenguajes de programación tienen un conjunto mínimo de instrucciones que permite especificar el control del algoritmo que se quiere implementar

Debe contener como mínimo de: selección e iteración.

Estas estructuras sirven para modificar el flujo de ejecución de las instrucciones.

CLASIFICACION




**Selección simple:** ejecuta determinadas instrucciones dependiendo de la toma de una decisión que se hace en función de una o varias condiciones.

**Selección múltiple**: puede ocurrir que en un problema real sea necesario elegir una entre varias alternativas en función del problema a resolver. Permite realizar distintas acciones (o bloque de acciones) **dependiendo del valor de una variable de tipo ordinal.**

***Estructuras de control iterativas:*** puede ocurrir que se desee ejecutar un bloque de instrucciones desconociendo el número exacto de veces que se ejecutan. Clasificadas en pre y post condicionales.

**Iterativas pre condicionales:** se ejecutan dependiendo de la evaluación de la condición, primero evalúan la condición y si es verdadera se ejecuta el bloque de acciones. Dicho bloque se puede ejecutar 0,1 o más veces.

**Iterativas repetitiva**: quizá un bloque de instrucciones se desee ejecutar un número exacto de veces. Donde la variable de control debe ser de tipo ordinal (entero, boolean o char). NO debe modificarse dentro del lazo. Los decrementos, incrementos o testeos son implícitos.

**Iterativa post condicional**: primero ejecuta el bloque de acciones y luego evalúan la condición. El bloque se ejecutará 1 o más veces, pero SIEMPRE mínimamente una vez.

## Modularización

La tarea de modularización implica dividir un problema en partes. Se busca que cada parte realice una tarea simple y pueda resolverse de manera independiente a las otras tareas.

Al descomponer un problema se debe tener en cuenta que cada sub problema resuelva una parte “bien” simple, pueda resolverse independientemente, y la solución del sub problema debe combinarse para resolver el problema original. Cada módulo resuelve un sub problema particular (define conjunto de acciones y los datos necesarios).

Ventajas de la descomposición del problema:

Distribuir el trabajo, reutilización del código en el mismo o distinto problema, aumenta la legibilidad, favorece el mantenimiento correctivo, facilidad el crecimiento de los sistemas.

Resultado de la etapa de diseño: cuales son los módulos, objetivo de cada uno, datos propios de cada uno, datos compartidos con otros módulos, conjunto de acciones para alcanzar ese objetivo.

## Procedimientos y funciones

Se debe elegir el lenguaje de programación para escribir los algoritmos de cada módulo y la declaración de sus datos. Los lenguajes de programación ofrecen <> opciones para implementar la modularización.

Definición del módulo: ¿qué hace cuando se ejecuta?, encabezamiento, tipo de modulo, identificación, datos de comunicación, declaración de tipos, variables, sección de instrucciones ejecutables.

Invocación del módulo: ¿qué se hace cuando se quiere usar el modulo? Se debe conocer de qué manera se invoca al módulo para que ejecute sus acciones.

Pascal ofrece procedure y función. Los cuales tienen características comunes pero algunas particularidades determinan el más adecuado. Se diferencian en el encabezado, la forma de invocación y lugar donde retorna el flujo de control una vez ejecutado el modulo.

***Procedure*** puede retornar 0,1 o más valores. Puede ocurrir que un módulo Procedure contenga además otros módulos. Esto se lo denomina procedimiento con módulos anidados.

***Function*** retorna un único valor de tipo simple. Luego ejecutado el modulo, el flujo de control retorna a la misma instrucción donde fue invocado el modulo.

## Comunicación entre módulos

**Variable global:** declaración hecha en la sección de declaración del programa principal o sea fuera de todos los módulos del programa y podrá ser usada en el programa y en todos los módulos. Entonces podría utilizarse como comunicación entre el programa principal y los módulos.

Desventajas: posibilidad de perder integridad de los datos al modificar involuntariamente en un módulo alguna variable que luego debe usar otro modulo, dificultad durante la etapa de verificación, demasiadas variables en la sección de declaración, posibilidad de conflicto entre nombres, falta de especificación del tipo de comunicación entre módulos y el uso de memoria. Por lo tanto, no se recomienda el uso de variables locales.

**Variable local al módulo:** declaración hecha en un módulo particular y solo podrá ser usada por ese modulo. Si este módulo contiene a su vez otros módulos, entonces esa variable podría ser también usada por todos los módulos interiores, si está declarada previo a ellos.

Variable local al programa: declaración hecha antes de la sección de instrucciones ejecutables del programa y después de la declaración de los módulos del programa. Su uso de limita a la sección de instrucciones ejecutables.

Para resolver el problema de las variables globales se recomienda el ocultamiento de datos y los parámetros. El ocultamiento significa que los datos exclusivos de un módulo NO deben ser visibles o usados por otros módulos, los datos propios del módulo se declaran locales al módulo, y los datos propios del programa se declaran locales al programa.

El uso de parámetros significa que los datos compartidos se deben especificar como parámetros que se transmiten entre módulos. Los datos compartidos se declararán como parámetros.

***Parámetros:*** datos utilizados para la comunicación entre programa y los módulos, de una manera explícita. Un parámetro es una variable que representa un dato compartido entre módulos o entre un módulo y programa principal.

Por lo tanto, el dato compartido se especificará como un parámetro que se transmite entre los módulos.

Un dato puede ser de *entrada*, *salida* o *entrada y salida* al módulo.

PARAMETROS

POR VALOR: datos de entrada al módulo. Es un dato que significa que el modulo recibe una copia de un valor proveniente de otro modulo o del programa principal. Con ese D el modulo puede hacer operaciones y/o cálculos, pero fuera del módulo ese dato no reflejará cambios. Debe ser tratado como una variable local del módulo*, puede significar gran utilización de memoria.*

POR REFERENCIA: datos de salida, datos de entrada y salida al módulo. El dato contiene la dirección de memoria donde se encuentra la info compartida con otro modulo o programa que lo invoca. El modulo opera con la info que se encuentra en la dirección de memoria compartida y las modificaciones que se produzcan se reflejan los demás módulos que conocen esa dirección de memoria compartida. Operan directamente sobre la dirección de la variable original por lo tanto no requiere memoria adicional.

## Estructura de datos

Conjunto de variables que poseen relación lógica entre si y que se pueden reconocer como un todo, bajo un único nombre. Permite representar objetos del mundo real que son más complejos que un dato simple

CLASIFICACION:

SEGÚN ELEMENTOS

- *Homogénea*: todos los elementos que la componen pertenecen al mismo tipo de dato
- *Heterogénea*: los elementos que la componen son de <> tipo de dato.

CONCEPTO DE ALGOTIRMOS, DATOS Y PROBLEMAS

SEGÚN OCUPACION EN MEMORIA

- *Estática*: la cantidad de elementos que contiene no varía durante el tiempo de ejecución del programa. Entonces la cantidad de memoria que utiliza no varia
- *Dinámica*: la cantidad de elementos puede e variar por lo tanto puede variar la memoria ocupada en tiempo de ejecución

SEGÚN ACCESO

- *Directo*: se puede acceder a un elem. en particular sin necesidad de pasar por los otros.
- *Secuencial*: para llegar a un elemento, puede ser necesario pasar por sus otros elementos.

SEGÚN LINEALIDAD

- *Lineal*: cada elemento puede tener 0 o 1 predecesor y sucesor
- *No lineal*: cada elemento puede tener 0,1 o más sucesores y predecesores.

**ESTUCTURA DE DATOS TIPO *RECORD**

Permite agrupar un conjunto de campos, con igual o diferente tipo de dato, bajo un único nombre. Debo especificar el nombre de cada campo y tipo de dato, cada campo se puede referenciar individualmente*. Estructura heterogénea, de acceso directo, estática.*

Operaciones permitidas: asignación *:=* (solo valida si ambas variables son del mismo tipo de registro) y acceso a cada campo en particular **.**; cada campo tiene su tipo de dato por lo tanto cada campo hereda las operaciones de su tipo base.

**ESTRUCTURA DE DATOS *ARREGLO***

Colección de elementos que se guardan consecutivamente en la memoria y se pueden referenciar a través de índices. *Estructura homogénea, de acceso directo, estática y lineal.*

Estructura de datos que permite acceder a cada componente a través de índices, que indican la posición de cada componente dentro de una estructura de datos.

Hay arreglos unidimensionales (**vectores**) o bidimensionales(**matrices**).

Operaciones validas: asignación de contenido a un elemento, lectura/escritura, recorridos, cargar datos a un vector, agregar elementos al final, insertar elementos, borrar elementos, buscar un elemento.

OPERACIONES DE RECORRIDO: consiste en recorrer el vector de manera total o parcial para realizar algún proceso sobre sus elementos.

La operación de *recorrido total* implica analizar TODOS los elementos del vector

La operación de *recorrido parcial* implica analizar los elementos del vector HASTA encontrar aquel que cumple con lo pedido. Puede ocurrir que se recorra todo el vector

OPERACIÓN DE CARGA COMPLETA: consiste en guardar un elemento en cada posición del vector.

OPERACIÓN DE CARGA DE DATOS: consiste en incorporar un elemento a continuación de otro desde la posición inicial en forma consecutiva. La operación debe controlar que la

cantidad de elementos no supere la dimensión física. DimL: cant de elementos del vector, dimF: dim física del vector, espacio suficiente: verificar dimL<dimF, condición corte: el ingreso de datos debe tener un fin.

OPERACIÓN AGREGAR UN ELEMENTO AL FINAL: consiste en incorporar el elemento a continuación del ultimo ingresado, o sea en la posición siguiente a la indicada por la dimL(dimL+1). Hay que verificar que haya lugar en la estructura. Y aumentar dimL+1.

```
const
    dimF=100
type
  vector=array[1..dimF] of integer;
procedure agregar(elemento)
begin
  exito:=false
  if (dimL<dimF) then begin
    exito:=true;
    dimL:=dimL+1;
    vector[dimL]:=elemento;
 end;
end;
```

*1 Insertar elemento al final*

OPERACIÓN INSERTAR UN ELEMENTO:

incorpora el elemento en una poción determinada o de acuerdo a un orden impuesto. Hay ***2 opciones***: en *una posición determinada* (debo verificar que la posición sea válida y el espacio en el vector, abrir el vector, asignar el valor y aumentar dimL) o *manteniendo un orden determinado*

```
Procedure insertarPos(elemento)
Var i:integer;
Begin
  exito:=false;
 if( dimL<dimF) and ((pos>=1) and pos<= dimL)) then begin
  exito:=true;
  for i:=dimL downto pos do
  v[i+1]:=v[i];
v[pos]:=element;
dimL:=dimL+1;
end;
```

*2 Insertar en posicion*

```
Procedure InsertarElemOrd (var v: vector;  var dimL: indice;
                           elem : TipoElem; var exito: boolean);

var pos: indice;

Begin
   exito := false;
   if (dimL < dimF)
   then begin
          pos:= BuscarPosicion (elem, v, dimL);
          Insertar (v, dimL, pos, elem);
          exito := true;
        end;
end;
// requiere verificar espacio disponible, buscar la posicion
// correspondiente mantieniendo el orden y luego insertar
```

*3 Insertar en un vector ordenado*

```
Procedure InsertarElemOrd (var v: vector;  var dimL: indice; elem : TipoElem;
                           var exito: boolean);

  Function DeterminarPosicion ( x: integer;  v:Vector; dimL: Indice): Indice;
     var pos : Indice;
    begin
        pos:=1;
         while (pos<=dimL) and (x > v[pos]) do
          pos:=pos+1;
        DeterminarPosicion:= pos;
      end;
  Procedure Insertar (var v:vector; var dimL:Indice; pos: Indice; elem:integer);
    var j: indice;
    begin
        for j:= dimL downto pos do
          v [ j +1 ] := v [ j ] ;
        v [ pos ] := elem;
        dimL := dimL + 1;
    End;
var pos: indice;
Begin
   exito := false;
   if (dimL < dimF) then begin
                        pos:= DeterminarPosicion (elem, v, dimL);
                              Insertar (v, dimL, pos, elem);
                              exito := true;
                          end;
end;
```

*4 Insertar un elemento en una posición determinada*

OPERACIÓN BORRAR UN ELEMENTO: consiste en eliminar un elemento determinado, si es borrar de una posición determinada debo verificar que la posición sea válida. Si es eliminar un elemento determinado, hay que verificar que exista dicho elemento.

```
Procedure BorrarPos (var v: vector;
           var dimL: integer; pos: posicion;
           var exito: boolean );
 var i: integer;
 begin
  exito := false;
  if (pos >=1 and pos <= dimL)
     then begin
           exito := true;
           for i:= pos + 1  to   dimL  do
              v [ i - 1 ]  :=  v [ i ] ;
           dimL := dimL - 1 ;
          end;
 end;
```

*5 Eliminar de posición determinada*

```
Procedure BorrarElem (var v: vector;  var dimL: indice;
                      elem : integer;  var exito: boolean);
var pos: indice;

begin
  exito:= false;
  pos:= BuscarPosElem (elem, v, dimL);
  if pos <> 0 then begin
                    BorrarPosModif (v, dimL, pos);
                    exito:= true;
                   end;
end;
// para esta operacion primero requiero buscar el elemento
// y luego borrarlo
```

*6 Borrar un elemento determinado*

OPERACIÓN DE BUSQUEDA: proceso de ubicar información particular en una colección de datos. Hay que considerar los siguientes casos: los datos están almacenados *sin orden (búsqueda lineal)*, o están *ordenados con algún criterio (búsqueda secuencial optimizada o dicotómica).*

Secuencial: comienza desde el principio y se avanza por el vector uno por uno, la solución debería recorrer el vector y detenerse al encontrar el elemento.

```
Function BuscoPosElemOrd (x: integer; v:Vector; dimL: Indice): Indice;
  var pos : Indice;
  begin
    pos:=1;
    while (pos <= dimL) and (x > v[pos]) do
            pos:=pos+1;
    if ( pos > dimL ) or (x < v [pos]) then pos:=0;
    BuscoPosElemOrd:= pos;
  end;
```

*7 Búsqueda de arreglos ordenados*

Dicotómica: se aplica para elementos con orden, compara el valor buscado X con el ubicado en el medio del vector. Si el elemento ubicado al medio es = a X entonces la búsqueda termina, si no es el buscado, debe quedarse con la mitad del vector que conviene para seguir la búsqueda.

Aplicado cuando los elementos tienen orden. Caso contrario, debería ordenarse el vector previamente. Las comparaciones con (1+log2(dimL+1))/2. Cuando dimL crece el número de comparaciones es log2(dimL+1)/2.

```
Procedure  BusquedaBin ( var v: Vector; var j: Indice;
                                        dimL: Indice, x : TipoElem) ;
   Var pri, ult, medio :  Indice ;
   Begin
     j :=0 ;
     pri:= 1 ;
     ult:= dimL;
     medio := (pri + ult ) div 2 ;
     While  ( pri < = ult ) and ( x <> v [medio]) do begin
            If ( x < v [ medio ] ) then ult:= medio -1 ;
                                   else pri:= medio+1 ;
             medio := ( pri + ult ) div 2 ;
     end ;
     If pri < = ult then  j := medio
                          else   j := 0 ;
   End ;
```

*8 Búsqueda dicotómica*

Lineal: aplicado para elementos sin orden, excesivo tiempo, comparaciones dimL+1/2, es ineficiente si el arreglo crece.

```
Function BuscarPosElem (x:integer; v:vector; dimL: Indice): Indice;
var pos:Indice; exito: boolean;
Begin
 pos:=1;
 exito:= false;
 while (pos <= dimL) and (not exito) do
      if (x = v[pos]) then exito:= true
                      else pos:=pos+1;
 if (exito = false) then pos:=0;
 BuscarPosElem := pos;
end;
```

*9 Buscar sin orden*

Cuando trabajamos con vectores se debe considerar

DIMENSIÓN FÍSICA: especificada en el momento de la declaración y determina su ocupación de memoria. No variara durante la ejecución.

DIMENSIÓN LÓGICA: determina al cargar elementos del vector. Indica cantidad de posiciones de memoria ocupadas con contenidos cargados desde la posición inicial en forma consecutiva.

## Alocación en memoria

Hasta ahora se trabajó con la **alocación estática(stack)** de la memoria, donde las estructuras de datos vistas hasta ahora se almacenan estáticamente en la memoria física del ordenador. El espacio se reserva con anticipación y no cambia durante la ejecución del rograma. Permite una comprobación de tipos en tiempo de compilación. Pero uno de los inconvenientes es su rigidez, porque estas estructuras no pueden crecer o decrecer durante la ejecución del programa.

La **alocacion dinámica(heap)** son variables dinámicas o referenciadas. Los espacios de memoria asignados a las variables dinámicas se reservan y liberan durante la ejecución del programa. No hay espacio de memoria reservado. Su ventaja es la flexibilidad ya que las estructuras dinámicas pueden crecer o decrecer durante la ejecución del programa.

## Tipo de dato *puntero*

Es un tipo de variable usada para almacenar la dirección en memoria dinámica de otra variable, en lugar de un dato convencional. Mediante la variable de tipo puntero (en stack) se accede a esa otra variable, almacenada en la dirección de memoria dinámica señalada por el puntero. Es decir, el valor de la variable tipo puntero es una dirección de memoria.

Se dice que el puntero apunta o señala a la variable almacenada en la dirección de memoria que contiene el puntero.

Un puntero es un tipo de dato simple que contiene la dirección de otro dato. Los punteros pueden apuntar solamente a variables dinámicas, o sea datos almacenados en heap. Un puntero ocupa 4 bytes. Cada variable de tipo puntero puede apuntar a un único tipo de dato, puede apuntar a una variable de cualquier tipo, incluso estructurados.

El puntero siempre va a ocupar en memoria estaría 4 bytes, independientemente a lo que apunta. Un dato apuntado no tiene memoria asignada. Para poder emplear variables dinámicas es necesario usar el tipo de dato PUNTERO que permite referenciar nuevas posiciones de memoria que no han sido declaradas a priori y que se van a crear y destruir en tiempo de ejecución.

Operaciones permitidas: asignación de un valor a una variable puntero, asignación de valor al objeto referenciado por el puntero, acceso a la información del objeto referenciado por el puntero, operaciones de entrada y salida, operaciones de comparación, eliminar un objeto apuntado que no se necesita.

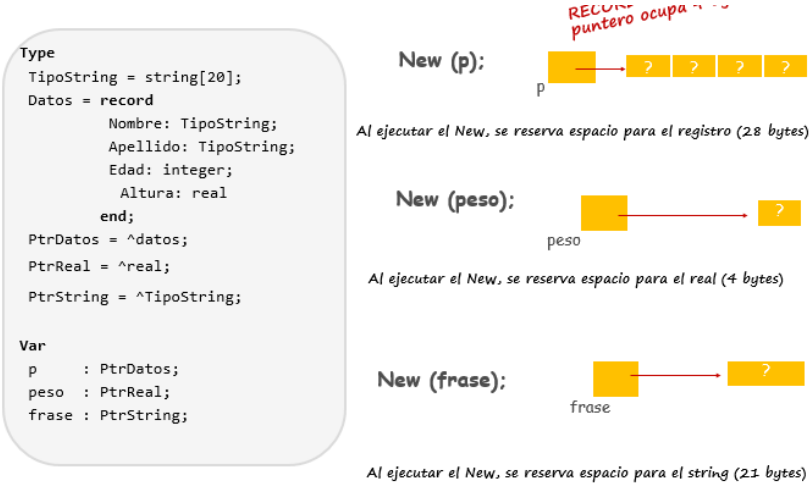
```
Type
 TipoString = string[20];
 Datos = record
           Nombre:TipoString;
           Apellido: TipoString;
           Edad: integer;
            Altura: real
         end;
  PtrDatos = ^datos;

Var
   p, q : PtrDatos;
```

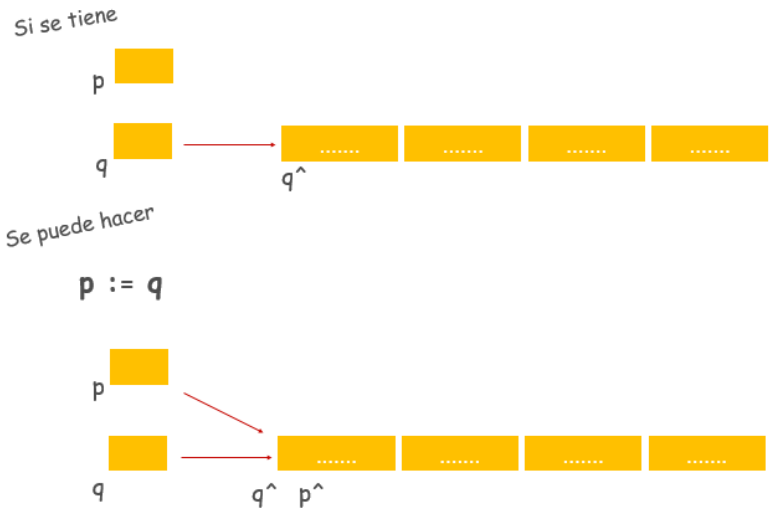
p := Nil;

q := Nil;

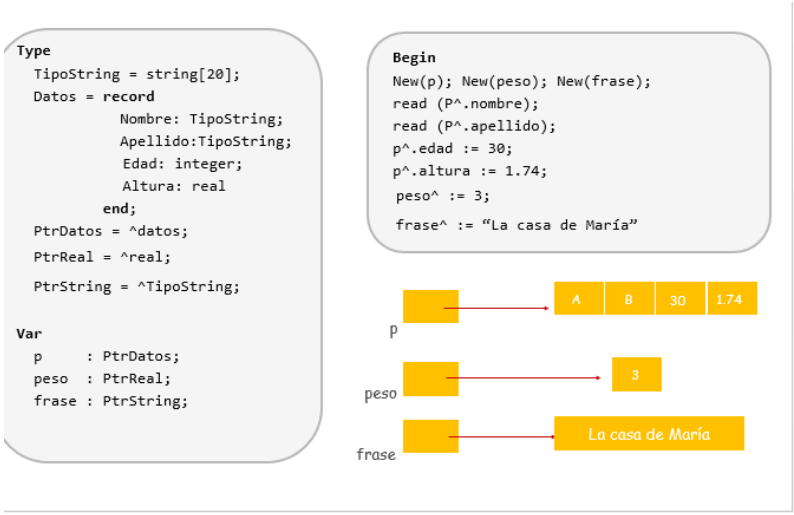

*10 Asignación nula a una variable puntero*




*11 Asignación de un valor a una variable puntero*




*12 Asignación de un valor a una variable puntero*

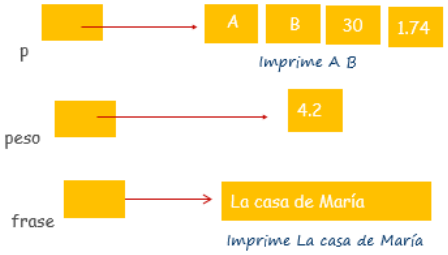



*13 Asignación de valor a la variable apuntada*

```
Type
 TipoString = string[20];
 Datos = record
          Nombre: TipoString;
          Apellido:TipoString;
          Edad: integer;
          Altura: real
         end;
 PtrDatos = ^datos;
 PtrReal = ^real;
 PtrString = ^TipoString;

Var
   p, q : PtrDatos;
   peso : PtrReal;
   frase : PtrString;
```

```
Begin
……..
write (P^.nombre);
write (P^.apellido);
peso^:=peso^ + 1.2;
write ( frase^);
```




*14 Asignación a la información de la variable referenciada*

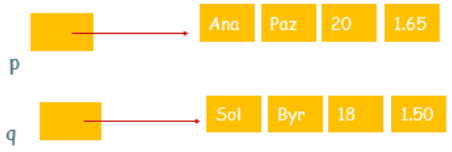
```
Type
 TipoString = string[20];
 Datos = record
          Nombre: TipoString;
          Apellido: TipoString;
          Edad: integer;
          Altura: real
         end;
 PtrDatos = ^datos;

Var
  p, q : PtrDatos;
```

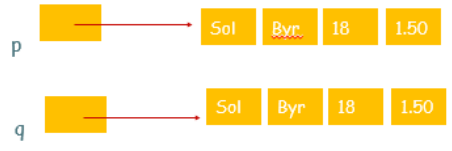

```
New (q);
Read(q^.nombre, q^.apellido, q^.edad, q^.altura);
p := q
```

*15 Asignación de punteros*

Si se tiene p y q como se muestra




y se hace **p^ := q^**




*( punteros distintos apuntando a valores iguales )*

*16 Asignación a la variable referenciada*

Operaciones de comparación: pueden aparecer en expresiones relacionales como p=q y p<>q

Operaciones de entrada/salida: no se pueden leer y/o escribir una variable puntero. Sí se puede leer y/o escribir los objetos que ellos referencian dependiendo del tipo apuntado.

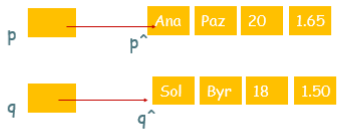
Si se tiene


Y se hace dispose (p); el efecto es que se "rompe" el enlace entre p y p^. No es posible volver a trabajar con el dato direccionado por p, por lo tanto, ese espacio de memoria puede ser "reutilizado".

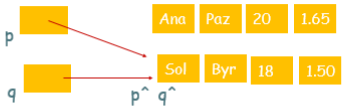

El contenido del puntero p queda indeterminado. No se lo puede utilizar a menos que se lo asigne nuevamente.

*17 Eliminación de una variable referenciada*

Si se tiene:




Y se hace **p := q;** el efecto es:

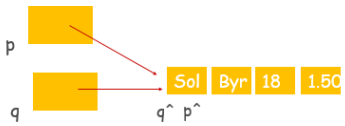



El espacio de memoria referenciado por p sigue "ocupado", pero no es posible referenciarlo.

Si en cambio se hace:

**Dispose (p);**
**p := q;**

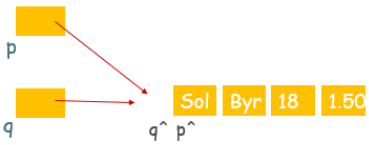
El efecto es:




Como el espacio de memoria referenciado por p fue "liberado", entonces puede ser reutilizado.

*18 Que ocurre cuando se usa el procedimiento dispose y cuando no se lo usa?*

Supongamos que:



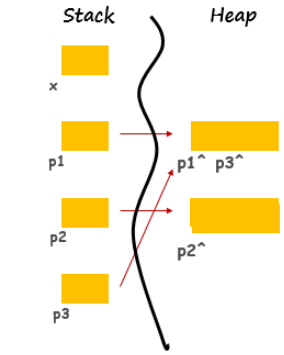

¿Qué ocurre si se hace

**Dispose (p)?**

El espacio de memoria referenciado por ese puntero será "liberado", por lo tanto, NINGÚN otro puntero que esté referenciando esa dirección podrá utilizarla.

```
Type
  pint= ^integer;

var  x : integer;
       p1, p2, p3: pint;
begin
 read (x) ;
 new (p1) ;
 new(p2) ;
 p1^ := x ;
 p2^ := p1^ + 1 ;
 read (x) ;
 p1^:= x ;
 p3 := p1 ;
 p1^ := p1^+p2^;
 writeln ('Elemento en p1: ', p1^);
 writeln ('Elemento en p2: ', p2^);
 writeln ('Elemento en p3: ', p3^);
End.
```




¿Qué pasa si...?

p1:=Nil?

Dispose (p2)?

Dispose (P1)?

## Tipo de dato *lista*

Colección de elementos *homogéneos*, con una *relación lineal* que los vincula (o sea que cada elemento tiene un único predecesor y sucesor). Los elementos que la componen no ocupan posiciones secuenciales o contiguas de memoria. Aparecen dispersos en la memoria, pero mantienen un orden lógico interno.

CARACTERISTICAS:

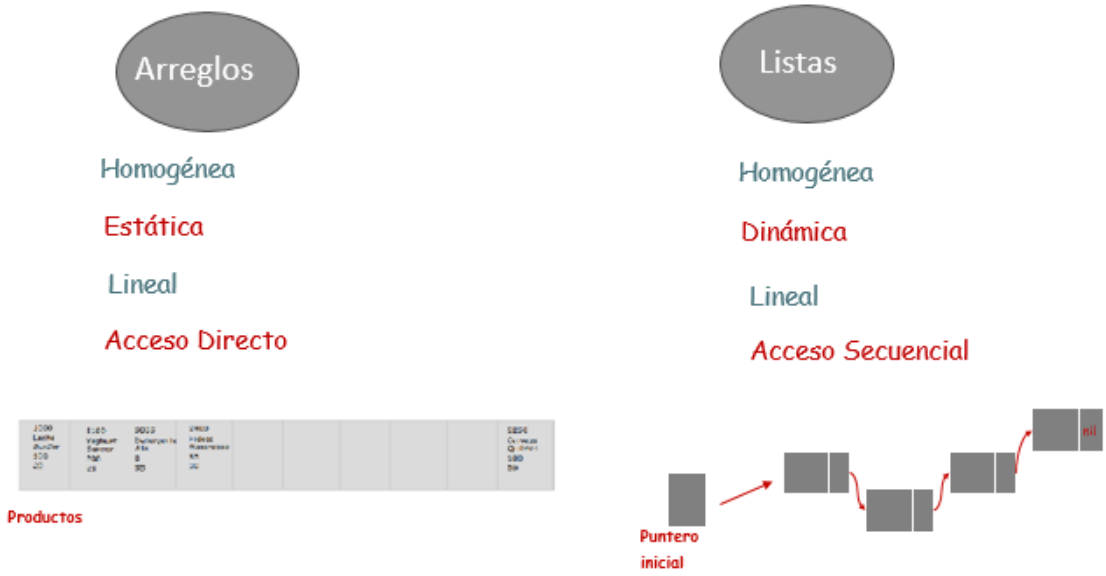
- Compuesto por nodos, los cuales se conectan por medio de enlaces o punteros.
- Cuando necesito agregar un nodo a la estructura, debo solicitar espacio adicional.

- Si ya no necesito un nodo, puede ser eliminado liberando memoria
- SIEMPRE se debe conocer la dirección del primer nodo de la lista para acceder a la información de la misma
- El ultimo nodo de la lista se caracteriza por tener su enlace en NIL

OPERACIONES VALIDAS: recorrer una lista, borrar un elemento, crear lista vacía, agregar un elemento al principio o final, buscar un elemento, insertar un nuevo elemento en una lista ordenada.

ANALISIS COMPARATIVO ENTRE LISTAS Y ARREGLOS:

| | Arreglos | Listas |
|---|---|---|
| Menoría | Almacenados en memoria estática | Almacenados en memoria dinámica |
| Ocupación | Ocupación de memoria resuelta en tiempo de compilación | Ocupación de memoria resuelta en ejecución |
| Posiciones | Ocupan posiciones consecutivas de memoria a partir de la posición inicial | Los elementos se disponen aleatoriamente en memoria y se relacionan lógicamente |
| Acceso | El acceso a un elemento es de forma directa | Debo recorrer desde principio hasta llegar a la posición del elemento buscado |




*19 Análisis comparativo de lista y arreglos*

**Espacio**: se refiere a la cantidad de memoria consumida por la estructura de datos dada. Si se supone igual cantidad de datos en las dos estructuras, se puede afirmar que: los vectores son más económicos y las listas requieren espacio extra para los enlaces.

Si no conozco la cantidad de datos que contendrá cada estructura, habría que haber un análisis

**Tiempo:** referido al tiempo que toma almacenar o recuperar datos. Se tendrá que analizar el tiempo que toma almacenar o recuperar datos, etc.

**Parámetros**: analizar cuál es el costo del pasaje de parámetros por valor y por referencia en arreglos o listas.

OPERACIONES

```
Type
  cadena50 = string[50];
  persona= record
        nom: cadena50;
             edad : integer
       end;
  lista= ^nodo;
  nodo = record
           datos : persona;
        sig  : lista;
         end ;
Procedure recorrido ( pri : lista);

Begin
 while (pri <> NIL) do begin
   write (pri^.datos.nom,
          pri^.datos.edad) ;
   pri:= pri^.sig
 end;
end;
```

*21 RECORRER UNA LISTA*

```
Type
 cadena50 = string[50];
  persona= record
        nom:cadena50;
            edad:integer
       end;
 lista = ^nodo;
 nodo = record
          datos : persona;
     sig  : lista;
        end;
function buscar (pri: lista; x:cadena50):boolean;
Var
   encontre : boolean;

begin
 encontre := false;
 while (pri <> NIL) and (not encontre) do
   if x = pri^.datos.nom  then
                        encontre:= true
                else
                        pri:= pri^.sig;
   buscar := encontre
End;
```

*20 BUSQUEDA DE ELEMENTO EN UNA LISTA*

```
Type
 cadena50 = string[50];
  persona= record
        nom:cadena50;
            edad:integer
       end;
 lista = ^nodo;
 nodo = record
          datos : persona;
     sig  : lista;
        end;
Procedure AgregarAdelante(var L:lista;
per:persona)
Var nue:Lista;
Begin
    New(nue);
    nue^.datos:=per;
    nue^.sig:=L;
    L:=nue;
End;
```

*22 AGREGAR ELEMETNO ADELANTE*

```
Type
 cadena50 = string[50];
  persona= record
        nom:cadena50;
            edad:integer
       end;
 lista = ^nodo;
 nodo = record
          datos : persona;
     sig  : lista;
        end;
procedure AgregarAlFinal2 (var pri, ult: lista; per: persona);
var  nue : lista;
begin
 new (nue);
 nue^.datos:= per;
 nue^.sig := NIL;
 if pri <> Nil then
                ult^.sig := nue;
              else
                pri := nue;
 ult := nue;
end;
```

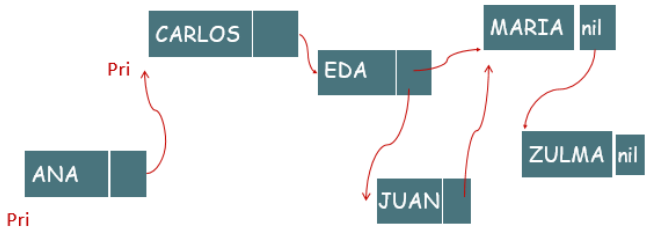
*23 AGREGAR ELEMENTO AL FINAL*

```
Type
  cadena50 = string[50];
   persona= record
          nom:cadena50;
              edad:integer
         end;
  lista = ^nodo;
  nodo = record
          datos : persona;
      sig  : lista;
         end;
Procedure BorrarElemento (var pri:lista; nom:cadena50; var exito: boolean);
var ant, act: lista;
begin
  exito := false;
  act := pri;
  {Recorro mientras no se termine la lista y no encuentre el elemento}
  while  (act <> NIL)  and (act^.datos.nom <> nom) do begin
      ant := act;
      act := act^.sig
  end;
  if (act <> NIL) then begin
                    exito := true;
                if (act = pri) then  pri := act^.sig;
                                     else  ant^.sig:= act^.sig;
              dispose (act);
          end;
 end;
```

*24 BORRAR UN ELEMENTO DE LA LISTA*

## INSERTAR UN ELEMENTO EN UNA LISTA

Supongamos que tenemos una lista de personas ordenadas alfabéticamente y queremos insertar los nombres Ana, Zulma y Juan.




Pasos a seguir:

1. Pedir espacio en memoria para el nuevo nodo
2. Guardar los datos en el nodo
3. Buscar posición donde se debe insertar (secuencialmente a partir del puntero inicial)
4. Reacomodar punteros. Considerar los tres casos:
   a. El nuevo elemento va en el inicio de la lista.
   b. El nuevo elemento va en el medio de dos existentes.
   c. El nuevo elemento va al final de la lista.

```
Type
  cadena50 = string[50];
   persona= record
          nom:cadena50;
              edad:integer
         end;
  lista = ^nodo;
  nodo = record
          datos : persona;
       sig  : lista;
         end;
Procedure InsertarElemento ( var pri: lista; per: persona);
var ant, nue, act: lista;
begin
  new (nue);
  nue^.datos := per;
  act := pri;
  ant := pri;
{Recorro mientras no se termine la lista y no encuentro la posición correcta}
  while (act<>NIL) and (act^.datos.nombre < per.nombre) do begin
      ant := act;
      act := act^.sig ;
  end;
  if (ant = act)  then pri := nue   {el dato va al principio}
                  else  ant^.sig  := nue; {va entre otros dos o al final}
  nue^.sig := act ;
end;
```

*25 INSERTAR UN ELEMENTO EN LA LISTA*

## Corrección y eficiencia

Corrección: técnicas para medir la corrección de un programa

Eficiencia: métodos para medir la eficiencia de un programa

Cuando debemos tomar decisiones, existe siempre una serie de factores que se deben analizar y buscar optimizar. En algunos casos, debemos buscar optimizar algunos en pos de sacrificar otros, según contexto y necesidades de la situación

CALIDAD DE UN PROGRAMA:

- *Corrección* ¿hace lo que se pide?: es el grado en que una aplicación satisface sus especificaciones y consigue los objetivos esperados por el cliente
- *Fiabilidad* ¿lo hace de forma fiable todo el tiempo? Grado que se puede esperar que una aplicación lleve a cabo las operaciones especificadas y con la precisión requerida
- *Eficiencia* ¿Qué RR de hardware y software necesito?: cantidad de RR de hardware y software que necesita una aplicación para realizar las operaciones con los tiempos de respuesta adecuados
- *Integridad* ¿puedo controlar su uso?: grado con que puede controlarse el acceso al software o a los datos a personal no autorizado
- *Facilidad de uso* ¿es fácil y cómodo de manejar? El esfuerzo requerido para aprender el manejo de una aplicación, trabajar con ella, introducir datos y conseguir resultados
- *Facilidad de mantenimiento* ¿puedo localizar los fallos?: esfuerzo requerido para localizar y reparar errores. Vinculado con la modularización y con cuestiones de legibilidad y documentación.
- *Flexibilidad* ¿puedo añadir nuevas opciones?: esfuerzo requerido para modificar una aplicación en funcionamiento
- *Facilidad de prueba* ¿puedo probar todas las opciones? Esfuerzo requerido para probar una aplicación de forma que cumpla con lo especificado en los requisitos

- *Portabilidad* ¿podré usarlo en otra máquina?: esfuerzo para transferir la aplicación a otro hardware o sistema operativo
- *Reusabilidad* ¿podré utilizar alguna parte del software en otra aplicación?: grado en que las partes de una aplicación pueden usarse en otras aplicaciones
- *Interoperabilidad* ¿podrá comunicarse con otras aplicaciones o sistemas informáticos?: esfuerzo necesario para comunicar con otras aplicaciones o sistemas informáticos
- *Legibilidad*, documentación: el código fuente de un programa debe ser fácil de leer y entender. Obliga a acompañar a las instrucciones con comentarios adecuados. Relacionado con la presentación de documentación. Todo el proceso de análisis y diseño del problema y su solución debe estar documentado mediante texto y/o gráficos para favorecer la comprensión, modificación y adaptación a nuevas funciones.

Un programa bien documentado será fácil de leer y mantener.

Se aconseja insertar comentarios, los identificadores se deberán elegir de forma auto explicativa, es recomendable hacer un comentario general del objetivo del programa.

Al hacer un mantenimiento, no solo se actualiza el código sino también los comentarios del programa

Los comentarios intercalados deben hacerse con criterio para contribuir a la claridad del programa.

Una vez escrita una posible solución es necesario verificar que cumple con el objetivo propuesto, conocido como *corrección del programa.*

Un programa es **correcto** si cumple con las especificaciones del problema a resolver. Por esta razón, es que la especificación debe ser completa, precisa y no ambigua.

Para medir la corrección de un programa, se usan distintas técnicas ***testing, walkthrough*** y ***debugging.***

TECNICA DE TESTING

Proceso mediante el cual se proveen evidencias convincentes respecto a que el programa hace el trabajo esperado.

Las evidencias se proveen mediante

| | |
|---|---|
| Diseñar un plan de prueba | Poner atención en los casos limites |
| Decidir cuales aspectos del programa deben ser testados y encontrar datos de prueba para cada uno de estos aspectos | Diseñar casos de prueba sobre la base de lo que hace el programa y no de lo que escribió del programa |
| Determinar el resultado que se espera que el programa produzca para cada caso de prueba | Mejor aún, diseñar casas de prueba antes de que comience la escritura del programa. Esto asegura que las pruebas no están pensadas a favor del que escribió el programa |

Cuando se tiene el plan de pruebas y el programa, el plan debe aplicarse sistemáticamente.

CONCEPTO DE ALGOTIRMOS, DATOS Y PROBLEMAS

*Pre-condiciones:* es la información que se conoce cómo verdadera antes de iniciar el programa o módulo.

*Post-condiciones:* es la información que debería ser verdadera al concluir el programa (o módulo) si se cumple adecuadamente los pasos especificados

Durante este proceso es importante analizar las pos condiciones en función de las precondiciones establecidas en el programa

Las precondiciones junto con las pos condiciones, permiten describir la función que realiza un programa, sin especificar un algoritmo determinado.

TECNICA DE DEBUGGING

Proceso mediante el cual se pueden identificar y corregir errores.

Puede involucrar: el diseño y aplicación de pruebas adicionales para ubicar y conocer la naturaleza del error. El agregado de sentencias adicionales en el programa para poder monitorear su comportamiento más cercano.

Los errores pueden provenir de varios caminos, por ejemplo: el diseño del programa puede ser defectuoso, el programa puede usar un algoritmo defectuoso.

TECNICA DE WALKTHROUGHS

Consiste en recorrer el programa ante una audiencia. La lectura del programa a alguna otra persona provee un buen medio para detectar errores. Esta persona no comparte preconceptos y está dispuesta a descubrir errores u omisiones. A menudo, cuando no se detecta un error, el programador trata de probar que no existe, pero mientras lo hace, puede detectar el error, o bien puede que el otro lo encuentre

EFICIENCIA DE PROGRAMAS

Una vez que se encuentra con una solución correcta es necesario medir cuantos recursos se utilizan. En particular aquí se analizan: **tiempo de ejecución** y **memoria utilizada.**

Para cada problema se pueden tener varias soluciones, algorítmicas correctas. Sin embargo, el uso de recursos de cada una de esas soluciones puede ser diferente.

***EFICIENCIA*** métrica de calidad de los algoritmos, asociada con una utilización óptima de los recursos del sistema de cómputo donde se ejecutará el programa, principalmente la memoria utilizada y el tiempo de ejecución empleado.

Si quiero optimizar los RR de memoria y tiempo, debo preguntarme: *cómo calcular el espacio ocupado por un programa, cómo calcular el tiempo requerido por un programa*. En caso de ser necesario, podré analizar*: cómo reducir el espacio ocupado por un programa, cómo reducir el tiempo de ejecución de un programa*.

CONCEPTO DE ALGOTIRMOS, DATOS Y PROBLEMAS

MEDICION DEL TIEMPO DE EJECUCIÓN:

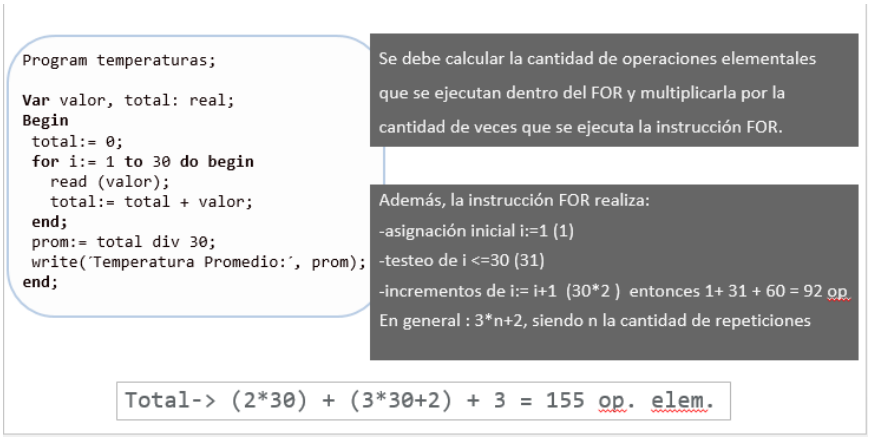
Depende de distintos factores, datos de entrada al programa (tamaño/contenido), calidad del código generado por el compilador utilizado, la naturaleza y rapidez de las instrucciones de maquina empleadas en la ejecución del programa, tiempo de algoritmo base

El tiempo de ejecución de un programa debe definirse como una función de la cantidad de datos de entrada. Para algunos programas, el tiempo de ejecución se refiere al tiempo de ejecución del "peor" caso. En estos casos, se obtiene una cota superior del tiempo de ejecución para cualquier entrada. Ejemplos: problema de búsqueda secuencial en vectores y lista. Puede calcularse de 2 maneras: **análisis empírico** o **teórico**.

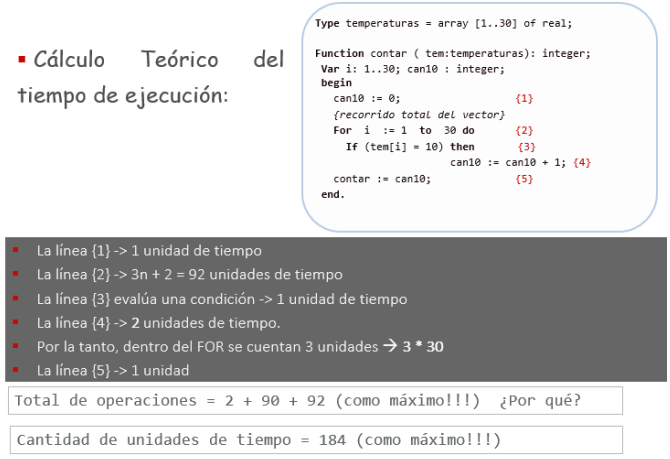
*Análisis empírico:* necesario ejecutar el programa y medir el tiempo empleado en la ejecución. Los inconvenientes de este análisis es que tiene varias limitaciones porque puede dar una información pobre de los RR consumidos: obtiene valores exactos para una máquina y unos datos determinados, completamente dependiente de la maquina donde se ejecuta, requiere implementar el algoritmo y ejecutarlo repetidas veces.

*Análisis teórico:* necesario establecer una medida intrínseca de la calidad de trabajo hecho por el algoritmo. Nos permite comparar algoritmos y seleccionar la mejor implementación. Lo positivo es que obtiene valores aproximados, aplicable en la etapa de diseño, independiente a la maquina donde se ejecute, permite analizar el comportamiento.

Para este análisis, se hace un cálculo teórico en el que: considera el número de operaciones elementales que emplea el algoritmo, considera que una operación elemental usa un tiempo constante para su ejecución, independiente al tipo de dato con el que trabaja, supone que cada operación elemental se ejecutara en una unidad de tiempo, supone que una operación elemental es una asignación, comparación u operación aritmética simple




*26 Regla FOR*



Aplicando la Regla del FOR, analicemos ahora el tiempo de ejecución del siguiente programa:

```
Program FA;
Var
  valor,i,j,suma :integer;
Begin
 suma:=0; {1}
 for j:= 1 to 300 do   {2}
  for i:= 1 to 200 do  {3}
    suma:= suma + I;   {4}
End.
```

Cantidad de operaciones (unidades de tiempo)

- La línea {1} -> 1
- La línea {2} -> 3n + 2 = 3*300 +2 = 902
- La línea {3} -> 3n + 2 = 3*200 +2 = 602
- La línea {4} -> 2

(((2 * 200 + 602) * 300) + 902) + 1= 301.503 ut

{4} * 200 + {3}

{4} * 200 + {3} * 300

{4} * 200 + {3} * 300 + {2}

{4} * 200 + {3} * 300 + {2} + {1}

*29 Regla for anidados*

Calcular la cantidad de operaciones elementales del siguiente programa

En el caso de una sentencia IF en su forma completa (then/else), debe calcularse la cantidad de operaciones que se realizan en cada parte y se debe elegir aquella que consuma mas tiempo → mayor cantidad de operaciones (el PEOR CASO).

```
Program uno;
Var
  valor, i, j, suma :integer;
 Begin
  read (valor);
  if (valor >8) then begin
                    suma:=0;
                    for i:= 1 to 3000 do
                        suma:= suma + I;
                  end
                  else begin
                    suma:=0;
                    for j:= 1 to 300 do
                       for i:= 1 to 200 do
                          suma:= suma + I;
                  end;
   end;
End.
```

((2*3000)+9002) + 1= 15.003 ut

(((2 * 200 + 602) * 300) + 902) + 1= 301.503 ut

Total de operaciones = 1 + 301.503= 301.504 (como máximo!!!) ¿Por qué?

*27 Regla if then/else*

Supongamos que el siguiente programa calcula la nota promedio de un alumno de Informática a partir de las notas obtenidas en sus exámenes finales. ¿Cuál es el tiempo de ejecución de la solución propuesta?

```
Program W;
Var
  nota,i,suma, prom :integer;
Begin
 suma:=0;  {1}
 total:= 0; {2}
 read (nota);
 while (nota<>-1) do begin {3}
   total:= total+1; {4}
   suma_N:= suma_N + nota; {5}
   read (nota);
 end;
 prom:= suma_N / total; {6}
 writeln (`Promedio: ´, prom)
End.
```

Se debe calcular la cantidad de operaciones elementales que se ejecutan dentro del WHILE y multiplicarla por la cantidad de veces que se ejecuta el WHILE. Como no se conoce esa cantidad se considera el PEOR CASO. Por ejemplo, se supone una cantidad de notas n…

Cantidad de operaciones (unidades de tiempo)

- La línea {1} -> 1
- La línea {2} -> 1
- La línea {3} -> n
- La línea {4} -> 2
- La línea {5} -> 2
- La línea {6} -> 2

((4 * n) + n+1) + 4 = (5n + 5) ut

*28 While/ repeat.. until*

Calculo del tiempo de ejecución en una solución modularizada: si se tiene un programa con módulos, es posible calcular el tiempo de ejecución de <> procesos, uno a la vez, partiendo de aquellos que no llaman a otros. Debe haber al menos un módulo con esa característica. Después puede evaluarse el tiempo de ejecución de los procesos que llamaron a los módulos anteriores y así sucesivamente.

Si un programa se va a utilizar algunas veces, el costo de su escritura y depuración es el dominante. Si un programa se va a ejecutar solo en entradas pequeñas, la velocidad de crecimiento puede ser menos importante. Un algoritmo eficiente pero complicado puede no ser apropiado para el mantenimiento por partes de un tercero